Accelerating 3D Non-Rigid Registration using Graphics Hardware

Nicolas Courty, Pierre Hellier

International Journal of Image and Graphics

Accelerating 3D Non-Rigid Registration using Graphics Hardware

Nicolas Courty[1]
[1] *UBS, Laboratoire Valoria*
Campus de Tohannic, 56000 Vannes, France
NICOLAS.COURTY@UNIV-UBS.FR

Pierre Hellier[2,3,4]
[2] *INRIA, Visages project, Campus de Beaulieu, 35042 Rennes cedex, France*
PIERRE.HELLIER@IRISA.FR

[3] *INSERM, Visages-U746, Campus de Beaulieu, 35042 Rennes cedex, France*

[4] *University of Rennes1 - CNRS, Campus de Beaulieu, 35042 Rennes cedex, France*

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

There is an increasing need for real-time implementation of $3D$ image analysis processes, especially in the context of image-guided surgery. Among the various image analysis tasks, non-rigid image registration is particularly needed and is also computationally prohibitive. This paper presents a GPU (Graphical Processing Unit) implementation of the popular Demons algorithm using a Gaussian recursive filtering. Acceleration of the classical method is mainly achieved by a new filtering scheme on GPU which could be reused in or extended to other applications and denotes a significant contribution to the GPU-based image processing domain. This implementation was able to perform a non-rigid registration of $3D$ MR volumes in less than one minute, which corresponds to an acceleration factor of 10 compared to the corresponding CPU implementation. This demonstrated the usefulness of such method in an intra-operative context.

Keywords: Non-rigid registration ; $3D$ image processing ; GPU implementation.

1. Introduction

In the last decade, it has become increasingly common to use image-guided navigating systems to assist surgical procedures [3]. The reported benefits are improved accuracy, reduced intervention time, improved quality of life, reduced morbidity, reduced intensive care and reduced hospital costs. Image-guided systems can help the surgeon plan the operation and provide accurate information about the patient's anatomy during the intervention. Image-guided systems are also useful for minimally invasive surgery, since the intraoperative images can be used interactively as a guide.

Current surgical procedures rely on complex preoperative planning, including various multimodal examinations: anatomical, vascular, functional explorations for brain surgery. Once all the information has been gathered and fused, it can be used for navigation in the operating room (OR) using image-guided surgery systems. To do so, a rigid registration of the patient's body with the preoperative data must first be performed . With an optical tracking system, and Light Emitting Diodes (LED), it is possible to track the patient's body, the microscope and the surgical instruments in real time. The matching of preoperative and intraoperative data is performed by identifying corresponding features (points or surfaces) in the two scenes.

Unfortunately, the assumption of a rigid registration between the patient's body and the preoperative images only holds at the beginning of the procedure. This is because soft tissues tend to deform during the intervention. This is a common problem in many image-guided interventions, the particular case of neurosurgical procedures can be considered as a representative case. When dealing with neurosurgery, this phenomenon is called "brain shift".

The magnitude of soft tissue deformation shows striking differences at each stage of surgery. Brain shift must be considered as a spatio-temporal phenomenon, and should be continuously estimated , or at least at key moments, to update the preoperative planning. To do so, intraoperative images (like intraoperative Magnetic Resonance Images or $3D$ ultrasound images) are acquired during surgery and can be used to estimate the deformation of soft tissues. Non-rigid image registration is then needed. Generally, a mono-modal registration method is preferred (i.e., registering a sequence of images of the same modality) since this problem is methodologically simpler.

There is a great amount of literature on non-rigid image registration, we thus refer the reader to comprehensive surveys on this domain [12,15,24]. Methods can be broadly classified according to the similarity measure that measures the discrepancy between the images to be registered and the type of regularization (the regularity of the estimated deformation field). Published methods are generally computationally expensive. Reported computation times vary between several minutes and several hours [16] and are in all cases incompatible with an application in the operating room.

This paper proposes a fast non-rigid registration method implemented on GPU and compatible with the image-guided surgery requirements. The contribution of the paper is twofold: firstly, $3D$ image processing is expressed as operations on $2D$ textures. Secondly, we propose an efficient recursive filtering scheme implemented on GPU that is shown to be 15 times more efficient than the software implemented version. The paper is organized as follow: after a short presentation of previous work on using the GPU in general purpose applications (section 2), we briefly recall some theoretical background on the used registration method (section 3), then our GPU implementation is proposed (section 4) along with some results (section 5) and a conclusion.

2. Related Work

Though first designed for the computer graphics industry, over the last few years graphics processing units have proven to be high performance computing platforms at low cost. With their increased programmability, it is now possible to consider execution of non-graphic applications on such boards. The principles of computation using a GPU exploit the capacities of the graphics pipeline that allow to transform geometric primitives like triangles into a collection of pixels (or fragments). This transformation can be parameterized through simple programs (also called fragment programs or shaders) that are executed on the GPU for every fragments of the primitive. Depending on the numbers of pipeline on the board, several fragments can be processed at the same time, leading to an implicit parallel execution of the transformation. In the context of non-graphics computation, it is thus possible to benefit from this implicit parallelism by defining a correct mapping between the problem and the geometric pipeline. Finally, since the data handled by the GPU are by nature four components vectors (Red/Green/Blue/Alpha), another opportunity for parallel execution of the program is given.

Several groups have explored these possibilities for a wide variety of computationally expensive problems (see [5] for a survey). Considering computer vision and image processing algorithms, it has been shown that GPU could speed up most of the classical filtering operations (convolution with simple filters like, for example, edge detectors) as well as compounded operations that require several filtering steps (motion estimation [21] for instance). This has lead to the development of several open source libraries like OpenVIDIA [4]. Let us note that in most of these implementations, filter kernels are relatively small (3×3 or 5×5). The overall GPU optimizations have been shown to decrease with the size of kernel because of the costly nature of the texture fetching operation (which is frequently the base for direct implementations of GPU image filtering).

Considering $3D$ image processing, some work has been done on visualization [10,11], segmentation [17], and filtering [9]. The use of graphics boards to speed-up medical applications has also drawn attention in the domain of tomography [23]. Regarding non-rigid registration methods, an approach using regularized gradient flow has been developed in [20] for the 2D case. In the case of volume registration, a non-linear warping of volumes with thin-plate splines have been exposed in [13] and the hardware acceleration through the use of 3D Bézier functions has been published in [18,19,7]. To our knowledge, our work constitutes the only attempt to implement a non-linear registration like the demons on commodity PC graphics boards. Among other, our recursive filtering method constitutes an efficient approach to volume processing on GPU, and exhibits better results for complete registration of 3D medical volumes.

3. 3D non-rigid registration method

3.1. *Overview*

We have chosen to use the Demon's method for its proved effectiveness and computational efficiency. In particular, this method has been shown to efficiently register an atlas toward a subject [1] and to register brains of different subjects [8]. The Demon's method was originally proposed by Thirion [22]. At each demon's location (usually the grid of demons is dense, i.e. every voxel is a demon), force are computed so as to repulse the model toward the data. The force depends on the polarity of the point (inside or outside the model), the image difference and gradients. Thirion has proposed the following expression for the deformation field:

$$u(s) = \frac{(I_1(s) - I_2(s))\nabla I_2(s)}{||\nabla I_2(s)||^2 + (I_1(s) - I_2(s))^2}$$

This amounts to a minmax problem: maximization of similarity and regularization of solution. For small displacements, it has been shown that the demon's method and optical flow are equivalent. The method alternates between the computation of forces and the regularization of the deformation field until convergence. Here is a synopsis of the algorithm:

```
DO
  1. Compute spatial and temporal gradients
  2. Compute dense grid of demons
  3. Regularize incremental deformation field
      using Gaussian filtering
  4. Update deformation field
  5. Interpolate deformed image
UNTIL CONVERGENCE
```

The convergence condition can be expressed according to the mean square error (MSE) between the reconstructed volume and the source volume. When the MSE decreases between two steps is less than a specified threshold, convergence is reached. In this paper, the five steps described above are implemented using GPU which leads to efficient implementation.

Computationally, the most costly steps are the computation of spatial gradients and the regularization of the deformation fields. These two operations basically consist in a convolution with a Gaussian filter for the regularization and with the first derivative of the Gaussian for the computation of spatial gradient. To optimize these two steps, we have chosen the recursive implementation of the Gaussian filter as proposed in [2].

3.2. *Recursive filtering*

The recursive Gaussian filtering makes it possible to compute infinite impulse filters with a bounded complexity. Deriche [2] proposes to approximate the Gaussian filter with 4^{th} order cosines-exponential functions as:

$$h_a(n) = (a_0 \cos(\frac{\omega_0}{\sigma}n) + a_1 \sin(\frac{\omega_0}{\sigma}n))e^{-\frac{b_0}{\sigma}n} + (c_0 \cos(\frac{\omega_1}{\sigma}n) + c_1 \sin(\frac{\omega_1}{\sigma}n))e^{-\frac{b_1}{\sigma}n}$$

It is shown that the approximation is fair for Gaussian filters with a standard deviation lower than 10 [2]. Separability is one of the most attractive features of the Gaussian filtering. Therefore, the three components of the deformation field will be processed successively. For a $1D$ signal x, the causal and anti-causal parts of the filtered signal y are expressed as:

$$y(i) = \sum_{k=0}^{k=3} b_k x(i-k) - \sum_{k=1}^{k=4} a_k y(i-k).$$

Numeric coefficients a_k and b_k are given in [2] for the Gaussian filter and its derivatives. In addition to [2], we propose in appendix the correct normalization of the filter.

The main advantage of recursive filtering is that the number of operations is bounded and independent of the standard deviation of the Gaussian filter. The latter is particularly appealing when filtering $3D$ images since a classical implementation is computationally expensive for large standard deviation. Let us finally note that this method allows one to minimize the number of texture fetching within the fragment program responsible for the filtering (which is one of the most time-consuming operations on the GPU).

4. Implementation

In this section, we present an original and efficient implementation of Thirion Demon's algorithm [22]. As presented in section 3.1, one iteration of convergence loop can be divided into five main operations. In our implementation, we factorized the two first steps (computation of temporal and spatial gradients and computation of demons in each voxel) in a unique step, followed by the regularization of the field (Gaussian filtering of the three field components) and finally the reconstruction of a final volume thanks to a trilinear interpolation of the current volume. The entire process is thus performed using GPU computation. We first concentrate on the representation of the volume in section 4.1. We then explain how to implement demons computations and trilinear interpolation on the GPU in section 4.2, and finally the recursive filtering scheme is described in section 4.3.

4.1. *Mapping the $3D$ volume on a $2D$ texture*

In our implementation, the data volume is not represented as a $3D$ texture, but instead as a big texture containing all the slices from the volume. This technique can be refered to as flat $3D$ texture, as first introduced by Harris [6]. There are several ways to decompose the data volume onto a plane. Figure 1 is an illustration of the

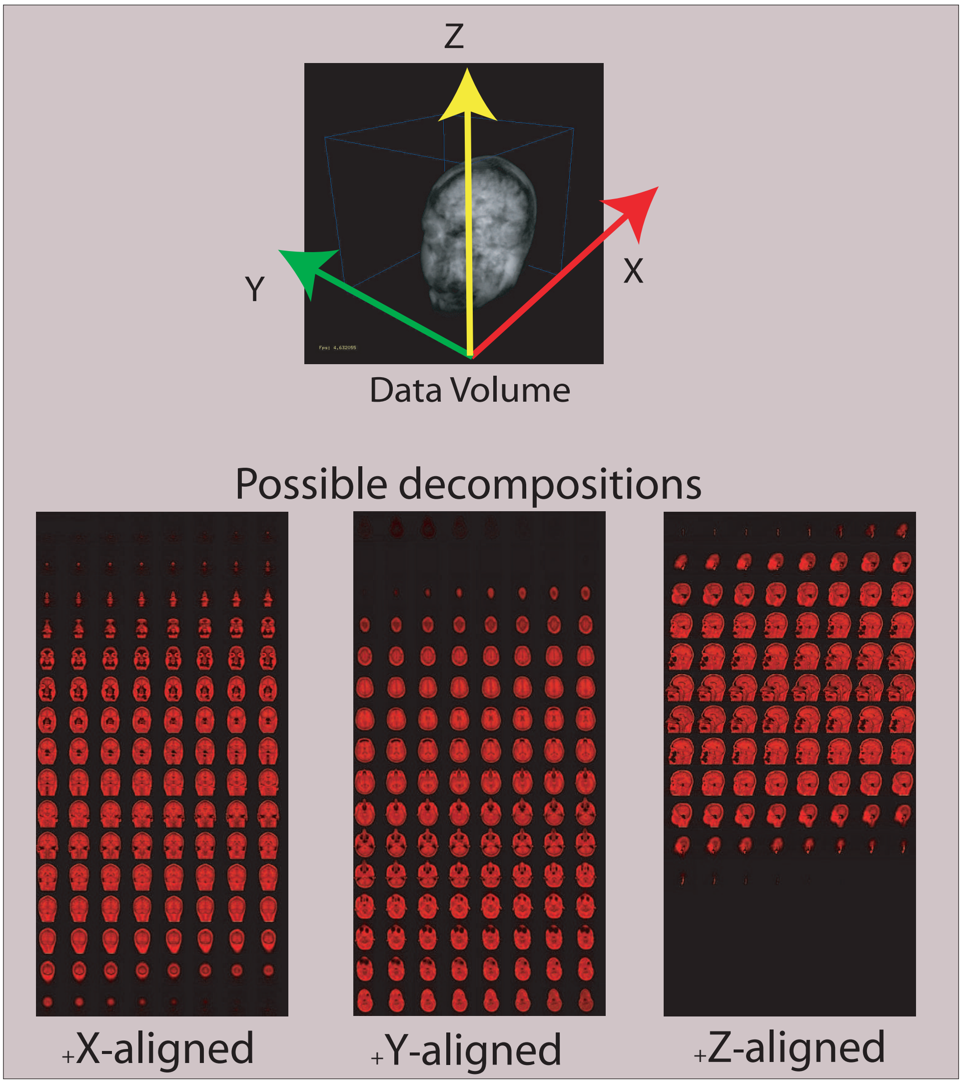


Figure 1. Possible decompositions of a $3D$ volume along three orthogonal axis. For each axis, two decompositions are possible if axial symetry is considered.

different possible decompositions along the three axes. Those different decompositions and the transformation operations from one to another will be considered in the filtering process for efficiency purposes.

There are two major advantages to such a decomposition: first of all, it is possible to process the entire volume in a single rendering pass, and a render-to-$3D$-texture extension is no longer needed. Once the whole volume has been uploaded to the graphics board memory, several processes can be performed without costly exchanges with the CPU memory (this assumption is true under the hypothesis that the volume can be contained entirely inside the board's memory, which is not often true, cf. discussion at the end of the paper). Secondly, accessing 2D textures in the video memory has been shown to be a much faster operation than accessing to 3D textures [6].

Nevertheless, this transformation is not straightforward. Two major problems arise: texture size limitation (4096×4096 for the current generation of graphic boards) and unicity of such a decomposition. Let us investigate the possible decomposition for a volume containing $D_x \times D_y \times D_z$ voxels, where D_k is the volume dimension along axis k. This $3D$ data should be mapped onto a $2D$ texture of size $N_i \times N_j$, under the constraint that $N_i \leq 2^{12}$ and $N_j \leq 2^{12}$ (this technical constraint will probably disappear in the future). Such a mapping might not be feasible, let us find the conditions under which it can be performed:

Let us first assume that the image planes are square dyadic images, i.e:

$$\exists p \in \mathbb{N} \quad \text{such that} \quad D_x = D_y = 2^p.$$

This assumption is not very restrictive when considering medical images, since this is very standard with actual scanners.

A solution to this problem can be seen as finding $n \in [1, p-1]$ such that:

$$N_j = D_z \times 2^{p-n} \quad \text{and} \quad N_i = 2^{p+n}$$

under the constraint $N_i \leq 2^{12}$and $N_j \leq 2^{12}$. D_z can be bounded as: $\exists k \in \mathbb{N} \quad \text{such that} \quad 2^{k-1} < D_z \leq 2^k$, what leads to:

$$p - n + k \leq 12 \quad \text{and} \quad p + n \leq 12$$

Therefore, adding and subtracting the two equations gives:

$$k \leq 24 - 2p \quad \text{and} \quad n \leq \frac{k}{2}$$

Since $n \in [1, p-1]$, the mapping can be performed if $D_z \leq 2^{24-2p}$. For instance, if $p = 8$ (i.e., $D_x = D_y = 256$), this amount to $D_z \leq 256$. Practically, this mapping is therefore feasible in most cases. From there, it is possible to access the whole volume in fragment programs by using a correct look-up function (given in Appendix B).

4.2. *Initialization and computation of demons*

The first step in the demon's algorithm implemented on a GPU consists in uploading the source volume and the destination volume decomposed along a particular axis. These two volumes are encoded in a unique texture (their elements are respectively stored in the red and green components of the texture elements, which limits texture fetching). Then the minimization loop is started until convergence (keeping in mind that the convergence is based on the MSE differential decrease to less than a threshold). The first step of this loop is the computation of "demons". They are expressed as a combination of the spatial and temporal gradients. It is hence possible to design a simple fragment program that computes spatial gradients and temporal gradients with finite differences and that will be executed on each element of the volume. This program is given as a reference in cg-like code in B.1.

The execution of this program on our volume outputs a texture containing an incremental displacement field. Two options are possible for regularization: either the incremental field is smoothed leading to a fluid regularization, or the total field is smooothed leading to an elastic regularization. This regularization is performed through the convolution of the displacement field with a Gaussian kernel. The GPU implementation of the recursive formulation of this filtering operation is detailed in the following paragraph.

4.3. *Recursive volume filtering on GPU*

The regularization of the field is the most critical part in the registration loop in terms of computational load. It consists of the Gaussian filtering of the three components of the deformation field. In order to speed up this process, the recursive filtering scheme presented in section 3.2 was implemented on the GPU. As stated previously, for each axis a *causal* and an *anti-causal* part has to be computed, which sums up to parsing the volume in one direction and then in the other direction. In order to factorize these two steps, the volume is transferred to the graphics board memory as a texture containing slices of two decompositions of the volume conduced along one axis and its opposite (on the red and green components of the texture), which allows for the handling of the causal and anti-causal parts at the same time.

To simulate the traversal process, we simply set up the view frustum to process one slice in a rendering pass. This process in then repeated for each slices along the given direction (which is equivalent to a full sweep of the volume). This amounts to the rendering of N rectangular quads where N is the number of slices in the considered direction. This fragment program can be found in B.2.

Figure 2 presents this transformation process for a given slice of the volume. Let us note that this method provides the interesting property of maximizing cache coherency along the volume sweeping (*i.e.* there is a greater probability that the needed volume element has already been accessed in the previous iteration and thus is present in the memory cache) .

This process outputs a texture containing both *causal* and *anti-causal* parts for a given axis. Since the process must be repeated in turn for the x, y and z axis, the volume needs to go through the two steps:

- addition of causal and anti-causal parts,
- re-orientation of the volume along the next axis

This step is done through a one-step rendering pass of the entire volume (*i.e.* rendering of a quad of the size of the texture containing the volume orientated along a given axis) with another fragment program (available in B.3).

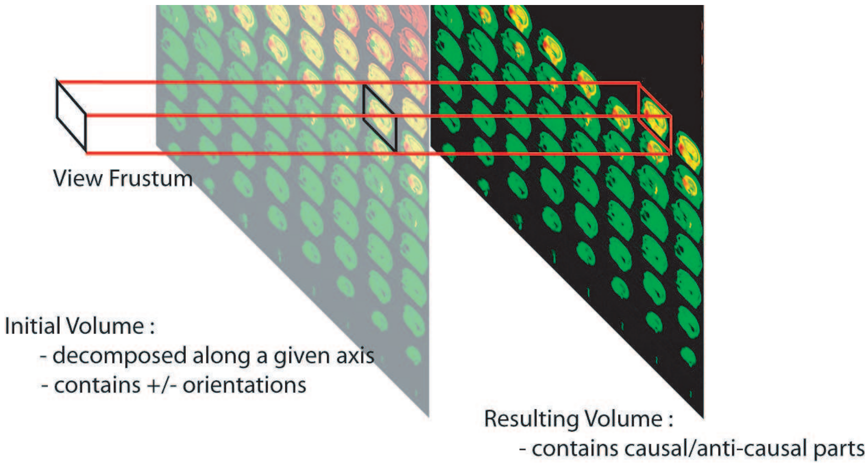


Figure 2. Sweeping the volume. To maximize computational efficiency, both causal and anti-causal parts are computed at the same time.

In this program, `fromVolumetoUVInvert` specifically allows one to get the anti-causal part from the volume. The whole process is described in Figure 3 which sums up the different parts of the filtering. Let us finally note that at the end of the third axis processing, a last addition/orientation step has to be performed to gather the final causal and anti-causal parts and transform the whole volume back in its original configuration. This ends the filtering process for one component of the displacement field.

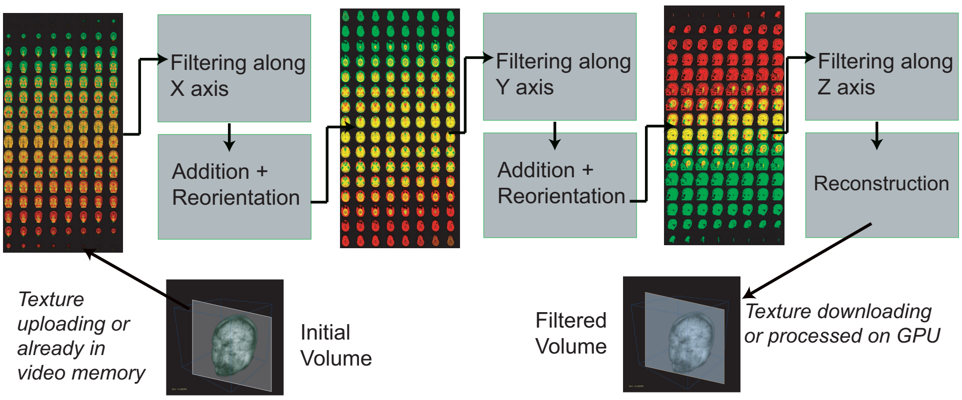


Figure 3. Different steps in the recursive filtering process. Boxes correspond to fragment shaders

4.4. *Trilinear interpolation*

Once the regularization has been done on the three axis, the initial volume is warped according to the computed displacement field before the next registration step. This corresponds to a Gauss-Newton resolution scheme where an incremental solution refines the previous one. This operation is performed through trilinear interpolation with a backward scheme. A simple fragment program is used to do all the computations (see B.4 for details).

The final volume is then reconstructed on the CPU. This last operation terminates an iteration of the demon algorithm.

5. Results

The registration method was tested on $3D$ magnetic resonance (MR) T1 images presented in Figure 4. The computer used for tests was an Athlon XP 2500+ equipped with a PCI-Express Quadro FX 1400 with 256 Mb of video memory. Because of the memory limitation, it was not possible to store all the data needed for the registration into video memory. Hence, the volumes were downsampled to a lower resolution, resulting in volumes of size 128^3 (the voxel resolution is $2mm$ isotropic). The volumes were first rigidly registered by maximization of mutual information [14].

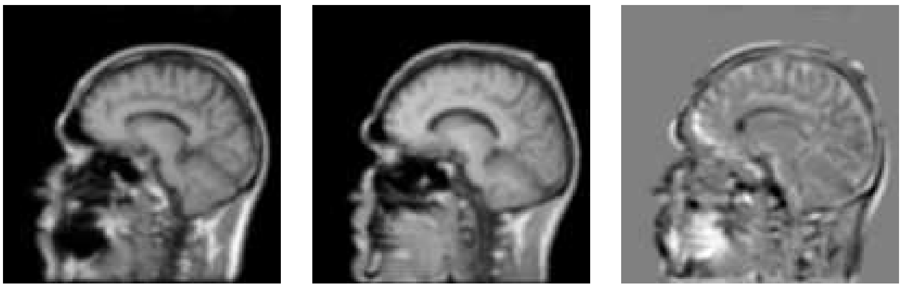


Figure 4. MR images used in the experiments. From left to right: the source volume, the target volume and the initial centered difference image. These images are $2D$ slices of 128^3 $3D$ data.

The registration method was tested with various regularization factors. Results of convergence and computation time are presented in Figures 5 and 6. The registration with a low regularization ($\sigma = 2$) was trapped in a local minimum, as assessed by the convergence rate of figure 5. As expected, increasing σ lead to smoother deformations and higher final discrepancy. This can be assessed visually in Figure 6 where final difference images are presented. Thanks to the recursive Gaussian filtering, the time per iteration does not depend on the standard deviation parameter and equals 2.2 seconds.

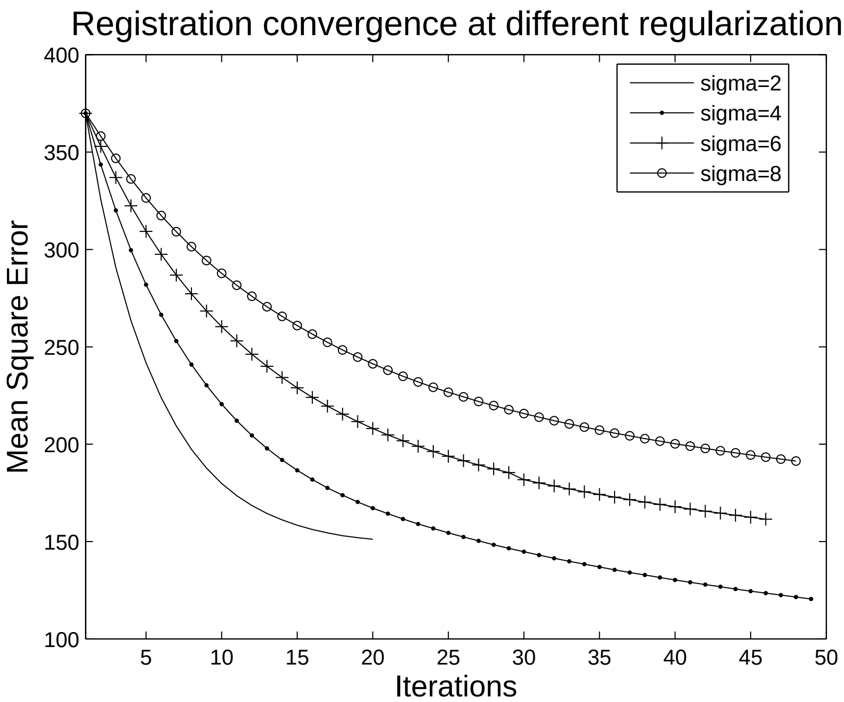


Figure 5. Convergence results: the figure plots the evolution of the Mean Square Error over iterations. The Mean Square Error decreases in all cases until convergence. The experiments were conducted for various regularizations. The experiment with a low regularization ($\sigma = 2$) converged to a local minimum. In all experiments the average time per iteration was 2.2 seconds, which clearly shows that the time per iteration is independent of the standard deviation of the filter

6. Conclusion

This paper presented a GPU implementation of a non-rigid $3D$ image registration method. To do so, $3D$ image processing tasks are expressed as operations on $2D$ textures and an efficient recursive filtering scheme implemented on GPU was used. The recursive filtering scheme is generic and can be used in many other applications such as image filtering, computation of image features (gradients, curvature, etc).

Results obtained on 128^3 volumes indicate that the GPU implementation is 10 times faster in average for comparable implementations and parameter tuning: we demonstrated that it is possible to perform a non-rigid registration in less than one minute. For a comparison, Noblet *et al.* [16] reported a computation time of 80 minutes for 50 iterations of the ITK demons method applied to 256^3 data. These results demonstrate the applicability of such methods in an image-guided surgery context.

In our implementation, the size of the volume remains a critical aspect, in that it was not possible to store all the data needed to process a 256^3 volume in video memory. We expect future generation boards to provide a sufficient amount of memory (512 Mo) to cope with this difficulty. It is also possible to decompose the input volume into smaller volumes that can be processed entirely on the GPU, thus still preserving acceleration from our method. This method would allow to efficiently treat larger next generation volumes (such as 512^3 or either 1024^3), and the evaluation of its performances is left as benchmarking perspectives for this work. Let us finally note that we expect a larger amount of pipelines to improve in a very significant way our algorithm.

Bibliography

1. B. Dawant, S. Hartmann, J-P. Thirion, F. Maes, D. Vandermeulen, and P. Demaerel. Automatic 3-D segmentation of internal structures of the head in MR images using a combination of similarity and free-form transformations: Part i, methodology and validation on normal subjects. *IEEE Transactions on Medical Imaging*, 18(10):909–916, 1999.
2. R. Deriche. Recursively implementing the gaussian and its derivatives. Technical Report 1893, INRIA, http://www.inria.fr/RRRT/RR-1893.html, April 1993.
3. N.L. Dorward. Neuronavigation - the surgeon's sextant. *Journal of neurosurgery*, 11(2):101–103, 1997.
4. J. Fung and S. Mann. OpenVIDIA: parallel GPU computer vision. In *ACM Multimedia*, pages 849–852, Singapore, November 2005.
5. GPGPU. General purpose computation on GPUs. http://www.gpgpu.org, 2006.
6. M. Harris, W. Baxter, T. Scheuermann, and A. Lastra. Simulation of cloud dynamics on graphics hardware. In *Proc. of the ACM SIGGRAPH/Eurographics Conf. on Graphics Hardware (EGGH-03)*, pages 92–101, Aire-la-ville, Switzerland, July 2003.
7. P. Hastreiter, C. Rezk-Salama, G. Soza, M. Bauer, G. Greiner, R. Fahlbusch, O. Ganslandt, and C. Nimsky. Strategies for brain shift evaluation. *Medical Image Analysis*, 8(4):447–464, 2004.
8. P. Hellier, C. Barillot, I. Corouge, B. Gibaud, G. Le Goualher, D.L. Collins, A. Evans, G. Malandain, N. Ayache, G.E. Christensen, and H.J. Johnson. Retrospective evaluation of inter-subject brain registration. *IEEE Transactions on Medical Imaging*, 22(9):1120–1130, 2003, in press.
9. M. Hopf and T. Ertl. Accelerating 3D Convolution using Graphics Hardware. In *Proc. of IEEE Visualization 1999*, pages 471–474, 1999.
10. J. Krueger and R. Westermann. Acceleration techniques for gpu-based volume rendering. In *Proc. of IEEE Visualization 2003*, 2003.
11. A. Lefohn, J. Kniss, C. Hansen, and R. Whitaker. Interactive deformation and visualization of level set surfaces using graphics hardware. *Proc. of IEEE Visualization 2003*, pages 75–82, October 2003.
12. H. Lester and S. Arridge. A survey of hierarchical non-linear medical image registration. *Pattern Recognition*, 32:129–149, 1999.
13. D. Levina, D. Deyc, and P. Slomka. Acceleration of 3d, nonlinear warping using standard video graphics hardware: implementation and initial validation. *Computerized Medical Imaging and Graphics*, 28:471–483, 2005.
14. F. Maes, A. Collignon, D. Vandermeulen, G. Marchal, and P. Suetens. Multimodality image registration by maximisation of mutual information. *IEEE Transactions on Medical Imaging*, 16(2):187–198, April 1997.

15. J. Maintz and MA. Viergever. A survey of medical image registration. *Medical Image Analysis*, 2(1):1–36, 1998.
16. V. Noblet, C. Heinrich, F. Heitz, and J.P. Armspach. Retrospective evaluation of a topology preserving non-rigid registration method. *Medical Image Analysis*, 10(3):366–384, June 2006.
17. A. Sherbondy, M. Houston, and S. Napel. Fast volume segmentation with simultaneous visualization using programmable graphics hardware. *Proc. of IEEE Visualization 2003*, 2003.
18. G. Soza, M. Bauer, P. Hastreiter, C. Nimsky, and G. Greiner. Non-rigid registration with use of hardware-based 3D Bézier functions. In *Proc. of Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, pages 549–556, 'Tokyo, Japan, November 2002.
19. G. Soza, P. Hastreiter, M. Bauer, C. Rezk-Salama, Ch. Nimsky, and G. Greiner. Intraoperative registration on standard pc graphics hardware. In *Proc. of the Bildverarbeitung für die Medizin (BVM)*, pages 334–337, 2002.
20. R. Strzodka, M. Droske, and M. Rumpf. Image registration by a regularized gradient flow - a streaming implementation in DX9 graphics hardware. *Computing*, 73(4):373–389, 2004.
21. R. Strzodka and C. Garbe. Real-time motion estimation and visualization on graphics cards. In *IEEE Visualization*, pages 545–552, Austin, Texas, US, October 2004.
22. JP. Thirion. Image matching as a diffusion process: an analogy with Maxwell's demons. *Medical Image Analysis*, 2(3):243–260, 1998.
23. F. Xu and K. Mueller. Accelerating popular tomographic reconstruction algorithms on commodity pc graphics hardware. *IEEE Tra. of Nuclear Science*, 52(3):654–663, 2005.
24. B. Zitova and J. Flusser. Image registration methods: a survey. *Image and Vision Computing*, 21:977–1000, 2003.

Photo and Bibliography

Nicolas Courty obtained in 1999 an engineer degree in computer science from Insa Rennes, and a Master degree from University of Rennes I (France). He obtained his phd degree from Insa Rennes in 2002 on the subject of image-based animation techniques. He then spent one year as post doctoral student in Porto Alegre, Brazil, working on GPU-based crowd simulation techniques. He finally integrated University of Bretagne Sud as assistant professor in 2004. His current research interests are the analysis/synthesis schemes in computer animation and simulation.

Pierre Hellier gratuated from "Supareo" (engineering school in aeronautics) in 1996 and obtained a master degree in applied mathematics from Toulouse University in 1996. He prepared his PhD on medical image registration at Inria Rennes and defended it in 2000. After one year as a postdoc at the Image Science Institute in Utrecht, the Netherlands, he was appointed as an Inria research scientist in 2001. He serves as regular reviewer for international conferences (CVPR, IPMI, MICCAI) and international journals (IEEE TMI, MedIA, IEEE EMB). He was program co-chair of the international conference "Medical Image Computing and Computer-Assisted Intervention" in 2004.

Appendix A. Normalization of Gaussian recursive filter

For this filter to be normalized, one has to compute:

$$\sum_{n=-\infty}^{\infty} h_a(n)$$

In [2], the following result is given:

$$\sum_{n=-\infty}^{\infty} h_a(n) = -a_0 \left(e^{\frac{2b_0}{\sigma}} - 1\right) \left(2\cos(\frac{\omega_0}{\sigma})e^{\frac{b_0}{\sigma}} - 1 - e^{\frac{2b_0}{\sigma}}\right)^{-1}$$

However, this needs to be corrected. Let us compute the sum on $\mathbb{R}^+$. Since the filter is symmetric, the final results will be obtained easily.

$$\begin{aligned} a\cos\left(\frac{\omega n}{\sigma}\right) + b\sin\left(\frac{\omega n}{\sigma}\right) &= \sqrt{a^2+b^2}\left[\cos\left(\frac{\omega n}{\sigma}\right)\sin(\theta) + \sin\left(\frac{\omega n}{\sigma}\right)\cos(\theta)\right] \\ &= \sqrt{a^2+b^2}\sin\left(\theta + \frac{\omega n}{\sigma}\right) \end{aligned}$$

with $\sin(\theta) = \frac{a}{\sqrt{a^2+b^2}}$ and $\cos(\theta) = \frac{b}{\sqrt{a^2+b^2}}$

Thus:

$$\begin{aligned} \sum_{n=0}^{\infty}\left(a\cos\left(\frac{\omega n}{\sigma}\right) + b\sin\left(\frac{\omega n}{\sigma}\right)\right)e^{-\frac{cn}{\sigma}} &= \sqrt{a^2+b^2}\mathcal{I}m\left(\sum_{n=0}^{\infty} e^{i(\theta+\frac{\omega n}{\sigma})}e^{-\frac{cn}{\sigma}}\right) \\ &= \sqrt{a^2+b^2}\mathcal{I}m\left(e^{i\theta}\sum_{n=0}^{\infty} e^{-\frac{cn}{\sigma}+i\frac{\omega n}{\sigma}}\right) \\ &= \sqrt{a^2+b^2}\mathcal{I}m\left(e^{i\theta}\sum_{n=0}^{\infty} z^n\right) \end{aligned}$$

With $z = e^{i\frac{\omega n}{\sigma}}e^{-\frac{cn}{\sigma}}$. z^n modulus goes to 0, which leads to:

$$\begin{aligned} \sum_{n=0}^{\infty}\left(a\cos\left(\frac{\omega n}{\sigma}\right) + b\sin\left(\frac{\omega n}{\sigma}\right)\right)e^{-\frac{cn}{\sigma}} &= \sqrt{a^2+b^2}\mathcal{I}m\left(\frac{e^{i\theta}}{1-z}\right) \\ &= \sqrt{a^2+b^2}\mathcal{I}m\left(\frac{e^{i\theta}}{1-e^{-\frac{cn}{\sigma}+i\frac{\omega n}{\sigma}}}\right) \end{aligned}$$

$$= \sqrt{a^2+b^2}\mathcal{I}m\left(\frac{\cos(\theta)+i\sin(\theta)}{1-e^{-\frac{c}{\sigma}}\cos(\frac{\omega}{\sigma}) - i\sin(\frac{\omega}{\sigma})e^{-\frac{c}{\sigma}}}\right)$$

$$= \sqrt{a^2+b^2}\mathcal{I}m\left(\frac{(\cos(\theta)+i\sin(\theta))\left(1-e^{-\frac{c}{\sigma}}\cos(\frac{\omega}{\sigma}) + i\sin(\frac{\omega}{\sigma})e^{-\frac{c}{\sigma}}\right)}{\left(1-e^{-\frac{c}{\sigma}}\cos(\frac{\omega}{\sigma})\right)^2 + \sin(\frac{\omega}{\sigma}e^{-\frac{c}{\sigma}})^2}\right)$$

$$= \frac{a\left(1 - e^{-\frac{c}{\sigma}}\cos(\frac{\omega}{\sigma})\right) + b\left(e^{-\frac{c}{\sigma}}\sin(\frac{\omega}{\sigma})\right)}{\left(1 - e^{-\frac{c}{\sigma}}\cos(\frac{\omega}{\sigma})\right)^2 + \left(\sin(\frac{\omega}{\sigma})e^{-\frac{c}{\sigma}}\right)^2}$$

Finally,

$$\sum_{n=0}^{\infty} h_a(n) = \frac{a_0\left(1 - e^{-\frac{b_0}{\sigma}}\cos(\frac{\omega_0}{\sigma})\right) + a_1\left(e^{-\frac{b_0}{\sigma}}\sin(\frac{\omega_0}{\sigma})\right)}{\left(1 - e^{-\frac{b_0}{\sigma}}\cos(\frac{\omega_0}{\sigma})\right)^2 + \left(\sin(\frac{\omega_0}{\sigma})e^{-\frac{b_0}{\sigma}}\right)^2}$$

$$+\frac{c_0\left(1 - e^{-\frac{b_1}{\sigma}}\cos(\frac{\omega_1}{\sigma})\right) + c_1\left(e^{-\frac{b_1}{\sigma}}\sin(\frac{\omega_1}{\sigma})\right)}{\left(1 - e^{-\frac{b_1}{\sigma}}\cos(\frac{\omega_1}{\sigma})\right)^2 + \left(\sin(\frac{\omega_1}{\sigma})e^{-\frac{b_1}{\sigma}}\right)^2}$$

Appendix B. CG-like code for the different fragment programs

We give as examples in this appendix the fragment programs that were used in this paper. The syntax is a cg-like syntax. We first begin by giving two look-up functions defined *wrt.* a decomposition along a given axis. The extension to other decompositions should be straightforward. Let us note $Zx = p - n$ and $Zy = p + n$ where p and n are defined in section 4.1. The look-up functions are given by:

```
float2 fromVolumeToUV(float3 coord){
    float X = fmod (coord.y,Zx);
    float Y = (coord.y - X)/Zx;
    return float2(X*Dx+coord.x+0.5,
                  Y*Dy+coord.z+0.5);}
```

```
float3 fromUVtoVolume(float2 uv){
    float x = fmod (uv.x,Dx);
    float z = fmod (uv.y,Dy);
    return float3(x, (uv.x-x)/Dx
                 +(uv.y-z)/Zy, z) ;}
```

B.1. *Demons computation*

Algorithm 1 Fragment program for computing demons

```
fragout_float main(vf30 In,
                   uniform texobjRECT source,
                   uniform float epsilon = 0.01){
    fragout_float O;
    float2 values = texRECT(source, In.TEX0.xy).xy;

    // compute spatial gradient
    float3 grad, posVol = fromUVtoVolume(In.TEX0.xy - float2(0.5,0.5));

    grad.x = texRECT(source, fromVolumeToUV(float3(posVol.x+1,posVol.y,posVol.z))).x -
             texRECT(source, fromVolumeToUV(float3(posVol.x-1,posVol.y,posVol.z))).x;
    grad.y = texRECT(source, fromVolumeToUV(float3(posVol.x,posVol.y+1,posVol.z))).x -
             texRECT(source, fromVolumeToUV(float3(posVol.x,posVol.y-1,posVol.z))).x;
    grad.z = texRECT(source, fromVolumeToUV(float3(posVol.x,posVol.y,posVol.z+1))).x -
             texRECT(source, fromVolumeToUV(float3(posVol.x,posVol.y,posVol.z-1))).x;
    grad = 0.5 * grad ;
    // compute temporal gradient
    float diff = values.x - values.y ; // source and target volumes are on the same texture
    // normalize
    float denom = diff*diff + grad.x*grad.x + grad.y*grad.y + grad.z*grad.z ;
    O.col.xyz = (denom<epsilon) ? float3(0,0,0) : grad * diff/denom;
    return O;}
```

B.2. *Recursive filtering*

Algorithm 2 Fragment program implementing recursive filtering along a given volume direction with a kernel given as input

```
fragout_float main(vf30 In,
                   uniform texobjRECT source,
                   uniform texobjRECT dest,
                   uniform float slice,
                   const uniform float kernel[16]} // kernel is the filter kernel{
   fragout_float O;
   float2 s = In.TEX0.xy;

   // current: non-modified volume, filtered: already computed values
   float2 current = texRECT(source,computeUV(s,slice));
   float2 lcurrent = texRECT(source,computeUV(s,slice-1));
   float2 llcurrent = texRECT(source,computeUV(s,slice-2));
   float2 lllcurrent = texRECT(source,computeUV(s,slice-3));

   float2 filtered = texRECT(dest,computeUV(s,slice-1));
   float2 lfiltered = texRECT(dest,computeUV(s,slice-2));
   float2 llfiltered = texRECT(dest,computeUV(s,slice-3));
   float2 lllfiltered = texRECT(dest,computeUV(s,slice-4));

   // both causal (O.x) and anti-causal part (O.y) are computed
   O.col.x = kernel[0]*current.x + kernel[1]*lcurrent.x +
             kernel[2]*llcurrent.x+ kernel[3]*lllcurrent.x -
             kernel[4]*filtered.x - kernel[5]*lfiltered.x -
             kernel[6]*llfiltered.x - kernel[7]*lllfiltered.x ;
   O.col.y = kernel[8]*current.y + kernel[9]*lcurrent.y +
             kernel[10]*llcurrent.y+ kernel[11]*lllcurrent.y -
             kernel[12]*filtered.y - kernel[13]*lfiltered.y -
             kernel[14]*llfiltered.y - kernel[15]*lllfiltered.y ;
   return O;}
```

B.3. *Reordering*

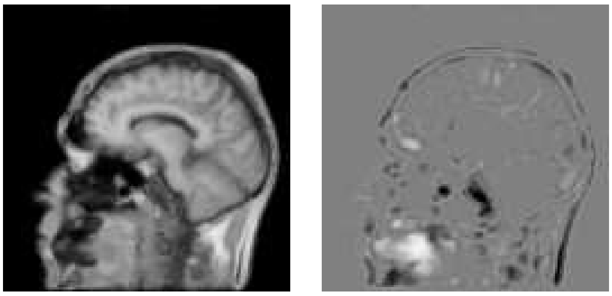
Algorithm 3 Fragment program that allows to process a volume before a new traversal

```
fragout_float main(vf30 In,
                   uniform texobjRECT source){
  fragout_float O;
  float3 tmp = fromUVtoVolume (In.TEX0.xy - float2(0.5,0.5));
  float2 s = fromVolumetoUV(tmp)+ float2(0.5,0.5);
  float2 invs = fromVolumetoUVInvert(tmp) + float2(0.5,0.5);
  O.col.x =  texRECT(source,s).x + texRECT(source,invs).y; // causal

  tmp.y = N_d - tmp.y ; // the opposite, with N_d the number of slices along the axis
  s = fromVolumetoUV(tmp)+ float2(0.5,0.5);
  invs = fromVolumetoUVInvert(tmp) + float2(0.5,0.5);
  O.col.y =  texRECT(source,s).x + texRECT(source,invs).y; // anticausal

  return O;}
```

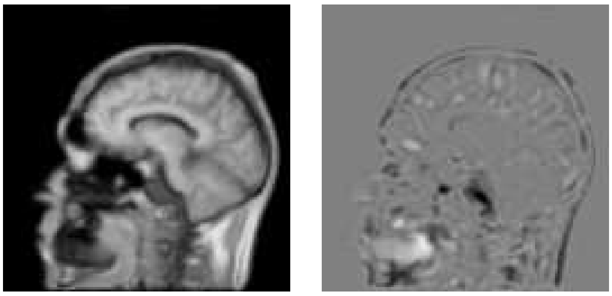
B.4. *Trilinear Interpolation*

Algorithm 4 Fragment program that performs trilinear interpolation

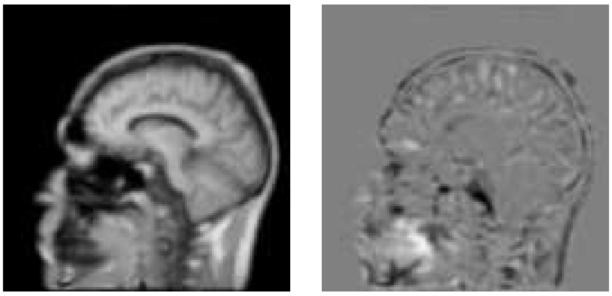
```
// performs tri linear interpolation of the volume
float volumeTrilerp(samplerRECT tex, float3 pos, float3 field){
  // we shall take integer value for the displacement
  float s1 = pos.x + floor(field.x);  float s2 = s1 + 1.0;
  float t1 = pos.y + floor(field.y);  float t2 = t1 + 1.0;
  float u1 = pos.z + floor(field.z);  float u2 = u1 + 1.0;

  float tex111 = texRECT(tex, fromVolumeToUV(float3(s1,t1,u1)));
  float tex121 = texRECT(tex, fromVolumeToUV(float3(s1,t2,u1)));
  float tex211 = texRECT(tex, fromVolumeToUV(float3(s2,t1,u1)));
  float tex221 = texRECT(tex, fromVolumeToUV(float3(s2,t2,u1)));
  float tex112 = texRECT(tex, fromVolumeToUV(float3(s1,t1,u2)));
  float tex122 = texRECT(tex, fromVolumeToUV(float3(s1,t2,u2)));
  float tex212 = texRECT(tex, fromVolumeToUV(float3(s2,t1,u2)));
  float tex222 = texRECT(tex, fromVolumeToUV(float3(s2,t2,u2)));

  float tx = field.x - floor(field.x);//interpolating factors
  float ty = field.y - floor(field.y);
  float tz = field.z - floor(field.z);

  float i1 = lerp(lerp(tex111,tex211,tx),lerp(tex121,tex221,tx),ty);
  float i2 = lerp(lerp(tex112,tex212,tx),lerp(tex122,tex222,tx),ty);
  return lerp(i1,i2,tz);}

fragout_float main(vf30 In,
                   uniform texobjRECT source){
  fragout_float O;
  float3 posVol = fromUVtoVolume(In.TEX0.xy-float2(0.5,0.5)); // position in volume
  O.col.x =  volumeTrilerp(source,posVol,f4texRECT(source,In.TEX0.xy).gba);
return O;}
```



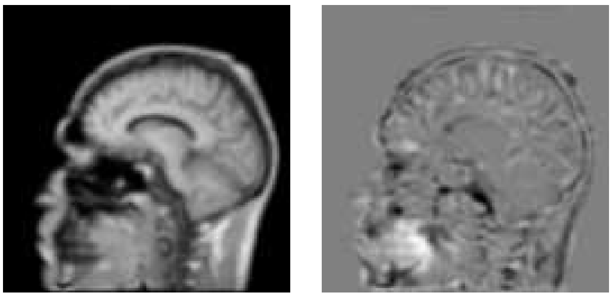
Results for $\sigma = 2$



Results for $\sigma = 4$



Results for $\sigma = 6$



Results for $\sigma = 8$

Figure 6. Final results of the registration for different regularization factors. Left: registered source volume to be compared with the target volume. Right: final discrepancy (difference image). The final discrepancy increases with the regularization. In all cases, volumes were correctly registered.